**UNIVERSITY OF CALIFORNIA,
IRVINE**

Semantic Software Engineering - A Survey with an Application

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Electrical and Computer Engineering

by

Adrianna Leung

Thesis Committee:
Professor Phillip Sheu, Chair
Professor Rainer Doemer
Professor Nader Bagherzadeh

2012

المنارة للاستشارات
www.manaraa.com

UMI Number: 1508197

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent on the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI 1508197

Copyright 2012 by ProQuest LLC.

All rights reserved. This edition of the work is protected against
unauthorized copying under Title 17, United States Code.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

المنارة للاستشارات

www.manaraa.com

© 2012 Adrianna Leung

للاستشارات المنارة

www.manaraa.com

DEDICATION

To

my mom

in recognition of her words of encouragement throughout the years

المنارة للاستشارات

www.manaraa.com

TABLE OF CONTENTS

	Page
LIST OF FIGURES	iv
LIST OF TABLES	v
ACKNOWLEDGMENT	vi
ABSTRACT OF THE THESIS	vii
CHAPTER 1	1
CHAPTER 2	11
CHAPTER 3	36
CHAPTER 4	60
REFERENCES	64
APPENDIX A	70
APPENDIX B	72

المنارة للاستشارات
www.manaraa.com

List of Figures

Figure 3.1	50
Figure 3.2	51
Figure 3.3	59

المنارة للاستشارات
www.manaraa.com

List of Tables

Table 1.1 3

Table 1.2 4

Table 1.3 6

المنارة للاستشارات

www.manaraa.com

Acknowledgment

I would like to express the deepest gratitude to my committee chair, Professor Phillip Sheu, who not only possesses tremendous knowledge in the area of semantic computing, but also the great enthusiasm in semantic software engineering research. Without his guidance and patience with my work schedule, this thesis would never have been complete.

I would also like to thank my committee members, Professor Rainer Doemer and Professor Nader Bagherzadeh for taking the time to review my thesis.

المنارة للاستشارات

www.manaraa.com

ABSTRACT OF THE THESIS

Semantic Software Engineering - A Survey with an Application

By

Adrianna Leung

Master of Science in Electrical and Computer Engineering

University of California, Irvine, 2012

Professor Phillip Sheu, Chair

As our modern software systems get larger and more complex, engineers constantly look for ways to improve software productivity. With the advent of semantic web, researchers start applying semantic computing technologies in software engineering. This thesis presents a survey of the state-of-art research in the field of semantic software engineering. In this thesis, I discuss the advantages of the various ontology-enabled approaches to software engineering, including the techniques proposed to automatically translate software requirements expressed in natural language to software designs and the various approaches to deriving semantics from software source code. I also introduce an interactive tool for software development, providing a means for software developers to create semantic annotations and improve the readability of software programs.

المنارة للاستشارات

www.manaraa.com

Chapter 1

According to Merriam-Webster dictionary, ontology is “a branch of metaphysics concerned with the nature and relations of being.” It is the science of describing various types of entities and how they are related. Today, the term ontology is also commonly defined as “specification of a conceptualization.” It is a formal, explicit description of the concepts and relationships that can exist for an agent or a community of agents. [1]
Ontology engineering, the study of methods and methodologies for building ontologies, has received enormous attention in the recent years, as many researchers recognize that the potential uses of ontologies are not limited in artificial intelligence. Not only does ontology provide a means for knowledge sharing, it also allows us to integrate heterogeneous, distributed information sources and reuse existing knowledge. A number of methodologies, such as DOGMA and ONTOCLEAN, have been proposed in the last decade to guide ontology builders to create ontologies that are not only highly usable, but are also reusable.

Contemporary ontology can generally be categorized according to its generality level. Upper, ontology (also known as top-level or fundamental ontology) refers to ontologies that can be used across all knowledge domains. Such ontologies describe only generic concepts and relations. Cyc and DOLCE are examples of upper ontologies. Domain ontologies, on the other hand, are only concerned with a specific domain. They aim to represent all entities and their relations with one another within a specific domain. Because they often specialize the terms of the top-level ontology, they are sometimes known as the lower-level ontologies.

www.manaraa.com

Meta-modeling and Ontology

A meta-model is “a precise definition of the constructs and rules needed for creating semantic models.” [2] Meta-models may be used:

- As a "schema" for semantic data that needs to be exchanged.
- As a language that supports a particular methodology or process
- As a language to express additional semantics of existing information.

Both meta-models and ontologies are used to represent relations between entities in a domain. A valid meta-model is ontology, yet ontologies may not necessarily be modeled explicitly as meta-models.

Ontology Languages

To facilitate interoperability between different agents, knowledge, represented in ontologies, are commonly encoded in ontology specification languages. The two widely-used ontology modeling paradigms are frame and description logic (DL). Frame-based languages, such as FLogic, are sometimes known as the traditional ontology language. A frame system is effectively a collection of related frames. We can think of a frame as a network of nodes and relations. There is a fixed “top levels” frame, followed by lower level terminals (also known as slots) that are filled by specific instances or data. Smaller “sub frames” must meet the conditions specified by their terminals.

Description logic, on the other hand, has emerged as a more popular paradigm in recent years, as the popularity of Semantic Web soars. Endorsed by World Wide Web Consortium (W3C), Web Ontology Language (OWL) and Resource Description

المنارة للاستشارات
www.manaraa.com

Framework (RDF) / RDF Schema are not only the standards used for the description of heterogeneous information across the web, they can also be used to link any information with semantics defined in an ontology. A RDF statement comprises an object, property, and value, commonly referred to as a "triple". A triple allows structured and semi-structured data to be shared across different domains. OWL, built on top of RDF, provides three increasingly expressive sublanguages. Of the three flavors, OWL-DL is the most popular choice amongst ontology engineers for its description logic reasoning capabilities. It was designed to support the existing Description Logic business segment and has desirable computational properties for reasoning systems. Like most other contemporary ontologies, OWL has the following components:

- An individual, also known as an instance in other languages, is an object in the domain of our interest. We can infer individuals as "instances of classes".
- A class is a concept, a collection, or a type of things. It is used to group a set of things with similar characteristics (i.e. properties). In taxonomy, a class may contain a collection of individuals, or it may also be a subclass of another class. Subclasses are subsumed by their superclasses.
- A property, or an attribute, describes a characteristic of a class or an instance. They are binary relation on the instances, and are used to link two individuals together. Properties are also known as roles in description logics and relations in UML.

While both OWL and RDF are essentially languages used to integrate heterogeneous and distributed data, OWL comes with a larger vocabulary and stronger syntax. It also provides more powerful machine reasoning than RDF.

	OWL Lite	**OWL DL**	**OWL Full**

المنارة للاستشارات
www.manaraa.com

Compatibility with RDF	No	No	All valid RDF documents are OWL full
Classes Descriptions	The only class description available in OWL lite is IntersectionOf	Classes can be described as UnionOf , ComplementOf, IntersectionOf, and enumeration	Classes can be described as UnionOf , ComplementOf, IntersectionOf, and enumeration

Table 1.1 Comparison of OWL Lite, OWL DL, and OWL Full

There are benefits to use either frame or DL. Frame is a good choice when one is trying to create ontologies for domains with closed-world semantics, while DL is the preferred approach when DL reasoning is needed. There are also research efforts made in combining OWL-DL and Flogic [3].

In addition to OWL, Semantic Web Rule Language (SWRL) is a rules-language, combining sublanguages of the OWL Web Ontology Language (OWL DL and Lite) with those of the Rule Markup Language (Unary/Binary Datalog). Since it is built on OWL DL, it shares the formal semantics and the rules are expressed in terms of OWL concepts, such as class and property. But while it may be more expressive than OWL DL, it has a disadvantage of undecidability. Therefore, it is recommended to express an ontology in OWL if possible, unless an additional expressive power is needed.

OWL	**Frame**
Consistency Checking	Constraint Checking
Open world assumption	Closed world assumption
Multiple model	Single model
Subclass relations can be inferred based on class definition	All subclass relations must be asserted explicitly

Table 1.2 Comparison of OWL and Frame

Query Languages

المنارة للاستشارات
www.manaraa.com

Now that we have described our data in an ontology language, how do we go about retrieving and manipulating the information expressed in an ontology language? SPARQL [4] is the de facto query language for RDF. A SPARQL query may contain triple patterns, conjunctions, disjunctions, and optional patterns. It only queries information represented in a RDF graph, which is a set of triples. There is no inference in SPARQL.

A commonly used query language for OWL is SPARQL-DL [5]. It uses SPARQL syntax and provides OWL-DL semantics for SPARQL basic graph patterns.

Alternatively, we may use Semantic Query-Enhanced Web Rule Language (SQWRL) [6] to retrieve knowledge from OWL ontologies. SQWRL is a SWRL based query language. It provides SQL-like operations that support negation as failure, disjunction, counting, and aggregation functionality. Similarly to SPARQL, in SQWRL we try to capture all concepts and relationships present in a pattern. Since SQWRL understands the semantics of OWL and SWRL rules, it understands not only the explicit, but also the inferred knowledge.

Ontology Editing Tools

There are a number of editing tools supporting the creation of ontologies. For example, Protege [7] from Stanford University is popular open source ontology editor and knowledge acquisition system that facilitate the modeling of ontology in Frames and in OWL.

Furthermore, Jena [8] is a Java framework for building Semantic Web applications. It provides extensive Java libraries for helping developers develop code that handles RDF,

المنارة للاستشارات

www.manaraa.com

RDFS, RDFa, OWL and SPARQL. Jena also comes with a rule-based inference engine to perform reasoning based on OWL and RDFS ontologies, and a variety of storage strategies to store RDF triples in memory or on disk.

Reasoners

A reasoner refers to a software application that is capable of inferring logical consequences from a set of asserted facts or axioms. It usually uses inference rules, expressed in the form of an ontology, and first-order predicate logic to perform reasoning. Either forward chaining or backward chaining may be used for inference. Some of the standard reasoning services include consistency checking, concept satisfiability, classification, and realization.

An example of an OWL reasoner is Pellet [9]. Pellet is a popular Java-based reasoner in the industry. Besides the basic reasoning functionalities, it also has support for SPARQL-DL conjunctive query answering, and incremental reasoning.

Other active reasoners include FaCT++ [10] and HermiT [11]. While HermiT is written in Java, it is based a different reasoning algorithm called hypertableau calculus, so it is theoretically faster. Unlike Pellet and HermiT, FaCT++ is developed in C++. Similar to Pellet, FaCT++ uses the optimized tableau algorithm. However, FaCT++ has no rule support. Both FaCT++ and HermiT are available as a Protégé plug-in, pre-installed for Protégé 4.x.

	Pellet	FaCT++	HermiT
Language	Java	C++	Java

المنارة للاستشارات
www.manaraa.com

Reasoning algorithm	Tableau	Tableau	Hypertableau
Rule support	Yes	No	Yes

Table 1.3 Comparison of Pellet, FaCT++, HermiT

1.1 Ontology and Software Engineering

In software engineering, an ontology "defines a set of representational primitives with which to model a domain of knowledge or discourse." [12] Ontology helps us better understand the structure of information among software agents. When knowledge is formally stored in a structured manner, terminological ambiguities can be greatly reduced. More importantly, we can easily perform analysis on the domain knowledge and potentially reuse. In [13], Pisanelli et al identified seven features of ontologies:

- an explicit semantic and taxonomy;
- a clear link between concepts, their relationships, and generic theories
- lack of polysemy within a formal context;
- context modularization;
- minimal axiomatization to pinpoint differences between similar concepts
- a good politic of name choice
- a rich documentation.

Numerous researchers proposed integrating ontologies in a number of disciplines in software engineering, such as:

- Software engineering methodologies. Software engineers can use ontology to represent software engineering methodologies such as the waterfall model.
- Requirements engineering

المنارة للاستشارات
www.manaraa.com

- Elicitation. Ontology can help us define complete, unambiguous, and consistent requirements.
- Requirement modeling. Ontology can be used to formally represent the requirements model, the acquisition structures for domain knowledge, as well as the knowledge of the application domain
- Requirement analysis. An ontology system may help us measure the quality of a requirements document and predict requirements changes in the future version of the document.

- Software design
  - Software modeling. Currently, the Model Driven Architecture-based languages have very little to no support in reasoning, so ontology can act as an extension to the current MDA approach.
  - Component-based software engineering. Similar to ontology engineering, the goal of CBSE is create a repository of reusable and independently modifiable software components. Researchers proposed using ontologies to define the semantics of components, along with the relations and communications between them.
  - Design patterns. While design patterns and ontologies are not identical, design patterns can be viewed as the more concrete description of solutions to specific problems in software design.
  - Programming languages and compilers. Researchers define general ontologies of programming languages with entities like identifiers, reserved words, and construct, in hope to translate knowledge bases from one language to another.
- Verification.

المنارة للاستشارات
www.manaraa.com

- Test case generation. Ontology can help generate basic test cases. Once our software designs are represented in ontologies, OWL provides a set of test cases that can be exploited to verify the functional requirements are met.
- Test case reuse. A test ontology allows us to better manage and retrieve existing test cases for reuse.

- Maintenance.

- Documentation. Ontology allows engineers to make use of software artifacts. Researchers actively propose ways to automatically populate ontologies from source code and documentations.
- Search. We can use SPARQL /SQWRL to look up relevant information about the software system stored in ontologies
- Bug tracking. A bug ontology model can give us insights into why certain modifications are made to the source code.
- Updating. Ontologies that captured details of code revisions can help us detect potential problems and investigate bugs in a system

Organization

The remainder of this thesis is structured as follows: In Chapter 2, I elaborate on the recent research activities in semantic software engineering, especially approaches to translate natural language requirement specifications into software designs and the techniques to derive semantics from source code. Chapter 3 presents my annotation tool that enables users to create semantic annotations of the source code. Finally, I

المنارة للاستشارات
www.manaraa.com

conclude this thesis with the summary and ideas for future work in Chapter 4.

المنارة للاستشارات
www.manaraa.com

Chapter 2

Semantic software engineering is the area of study that applies the semantic technologies in the field of software engineering. It is concerned with the representation of goals in formal specifications, sometimes known as software modeling, along with the extraction of meaning from formal languages.

Semantic Software Engineering

Most tasks in the software development lifecycle, especially requirements elicitation, software design, are difficult to automate because they involve a tremendous amount of domain knowledge. In [13], Verma et al. describe how a collection of semantic models, called semantic bus, may help to automate steps in the development process. By defining semantic representation in an ontology, tools used in different phases can communicate knowledge across phases.

In the previous chapter, we studied ontology, the backbone of semantic techologies. In this chapter, I will elaborate on the recent research activities in the field of semantic software engineering and look at how the ontology-enabled semantic technologies may improve the reusability, sharing and extensibility of software.

2.1 Requirement Engineering

Requirements Engineering (RE) is "the branch of software engineering concerned with real-world goals for, functions of, and constraints on software systems. It is also concerned with the relationship of these factors to precise specifications of software behavior, and to their evolution over time and across software families." [14]

المنارة للاستشارات
www.manaraa.com

A software requirements specification is the complete description of how a software system should behave and its other characteristics. Requirements in the specification can generally be categorized into functional and non-functional. Functional requirements define the behavior of a software application, or tasks that must be performed by a system. Non-functional requirements, on the other hand, are also known as the quantification requirements. Unlike functional requirements, quantitative requirements define a system's non-behavioral goals, such constraints, quality attributes, and quality goals. Requirements engineers are responsible for defining both functional and non-functional goals of all the stakeholders affected by a software system. Thus, a good communication of the requirements is very important. However, these goals may not always be explicit and can sometimes be difficult to articulate. A vast majority of requirement specifications (RSs) are written in the inherently ambiguous natural language (NL). Kormer has identified the following problems commonly seen in software requirements [15]:

- Nominalization
- Incompletely specified process words
- Nouns without reference index
- Incompletely specified conditions
- Modal verbs of necessity

Ontology in RE

In recent years, many research activities have been devoted to applying ontology in RE, hoping to overcome the problems mentioned above.

المنارة للاستشارات
www.manaraa.com

One of the most challenging activities in RE is requirements elicitation, in which analysts must gather all requirements and identify the domain constraints through meetings and interviews with the stakeholders. An application domain ontology can help maintain all information relevant to the application. Meanwhile, a quality ontology helps manage various qualities concerning the software, an application's non-behavioral attributes. Moreover, these two types of ontologies may serve as a reminder of all the necessary elicitation questions the requirements analysts need to ask or discuss with the stakeholders.

Modeling requirements and domain knowledge

Riechert et al. describe in [16] an ontology, known as SWORE, to support the RE process semantically. It defines the fundamental concepts in RE, such as Stakeholder and Abstract Requirement. Abstract requirements may have goals, scenarios, and requirements as subclasses. These subclasses have properties of defines and details, allowing us to model how they are interdependent on each other.

Dobson and Sawyer present in [17] an ontology that provides a common language to define dependability requirements. Unlike SWORE, it focuses on non-functional attributes such as availability, reliability, safety, integrity, maintainability, and confidentiality.

Al Balushi et al. [18] propose an ontology-enabled NFR tool, called ElicitO, to help capture precise and comprehensive quality requirements that represent constraints on functional requirements in elicitation interviews. ElicitO is based on two ontologies: a

المنارة للاستشارات
www.manaraa.com

quality ontology and a functional domain ontology. The quality ontology includes metrics, characteristics, and sub-characteristics defined in ISO/IEC 9126.

Soffer et al. [19] propose a framework to model off-the-shelf information system requirements (OISR). The ontology-based OISR model has four elements, namely business processes, business rules, information objects, and required system services. The framework uses ontology to support the evaluation of the available modeling language that may guide us in the selection, implementation and integration of purchased off-the-shelf information system.

Ontology with Use Case

In requirements engineering, a use case is used to graphically describe the interactions between an external actor (or a user) and a software system. It is a modeling approach that is often used to improve the understandability. A use case usually includes actors, tasks, extension positions, as well as preconditions and postconditions. As today's software systems become larger and more complex, we need a large collection of use cases to adequately specify all the different ways to use the system. Thus, it makes sense that we reuse use cases whenever possible. Retrieving a use case created in the past is, however, not a trivial task. Researchers proposed using ontology to annotate use cases with semantic information. In addition, the application domain and the system behavior ontology can support a smarter retrieval of use cases based on this semantic information.

In [20], Caralt and Kim describe an approach that uses ontology to augment use cases with semantic information for the easier and more accurate retrieval of use cases in the

المنارة للاستشارات
www.manaraa.com

future. It uses a subset of ResearchCyc, also known as ACTION, as well as WordNet for linguistic relationships. ACTION, combined with domain ontology and linguistic ontology (WordNet), can be used to resolve the ambiguities of the natural language in specifications.

Ontology with Goals and Scenarios

For many years, researchers look for ways to eliminate ambiguous words and incomplete constraints that often appear in requirements. Prior to designing and implementing software applications, models are often built to elaborate requirements and explore designs. Not only does modeling help facilitate communication between requirements engineers and the stakeholders, it also helps requirements engineers identify details that may be missed during the initial elicitation. For today's larger and more complex systems, however, creating software models is not a trivial task.
A number of approaches and methods have been proposed to automate the model extraction from requirements written in natural languages. Traditionally, there are two major approaches to modeling software requirements. Goal modeling is one of them. Goal models illustrate functional and non-functional goals and their impact on each other through AND/OR graphs. They help requirements engineers check for completeness and conflicts. However, they may sometimes be hard to elicit, or too abstract, covering only the classes of intended behaviors, leaving out important details. Shibaoka et al. [21] enhanced the goal modeling process with ontology called GOORE. GOORE supports the decomposition of goals during the requirement elicitation process. Given an input of a domain ontology and an incomplete goal graph, the content of the

المنارة للاستشارات
www.manaraa.com

goal descriptions expressed in natural language can be mapped into the words in the thesaurus part of the ontology. A set of inferred ontological concepts can then be added to the goal graph. Using inference rules expressed in Prolog, relationships between concepts can automatically be discovered as well. GOORE supports goal refinement and completeness checking.

Another approach to modeling is scenario-based. Requirement engineers sometimes represent the expected behavior of a software system with a set of agent interaction scenarios. Unlike the goal-oriented models, scenarios are more informal and more easily accessible to stakeholders. However, they are inherently partial and can only cover certain specific behaviors. They may also lead to premature design decisions about event sequencing and distribution of responsibilities among system agents.

Lee and Gandhi propose in [22] an Ontology-based Active Requirements Engineering framework that combines the modeling techniques such as goal-driven scenario composition, requirements domain model, and viewpoints hierarchy.

Semantic Wiki

The Semantic Web has garnered a lot of attention in the research community in the past decade. One of the most popular applications of the semantic web is the semantic wiki. A semantic wiki is an enhanced wiki that incorporates an underlying model of its page content knowledge. Unlike a traditional wiki, it allows users to add annotations and store interrelations between pages. Many also support ontology reasoning. With the rise of semantic technology, more researchers propose the use of semantic wiki for

المنارة للاستشارات
www.manaraa.com

requirement engineering. It helps engineers learn about the application domain and quality characteristics before they go elicit requirements.

In [23], Decker et al. suggested a new paradigm of Wikitology that supports the exchange of reusable specification documents. Specification documents are “self-organized”, given the nature of the wiki system. More importantly, the domain knowledge can easily be expanded, making reuse feasible. Using Wikitology, requirements engineers can easily link related and relevant documents together, offering suggestions for references. Moreover, they can use semantic annotations to define and check for consistencies.

Supporting Tools

In addition to modeling, ontology can also help us translate requirements into software design models. In [24], Verma et al. present a tool called Requirements Analysis Tool (RAT) that supports both a number of analysis on requirement specifications. It uses three types of user-defined glossaries (agent, action, and modal) to parse requirements documents and extract structured content. Given a requirement document, RAT outputs its structured content containing the requirement syntax type, all the identified agent, action, and modal word, along with different constituents of the conditional requirements. Domain specific knowledge can be defined using the requirements relationship glossary. The glossary lists a set of requirement classification classes, their super class, all the keywords used to identify the class, as well as the relationship between every class. With the extracted content, requirements can then be translated into a semantic graph, represented in OWL. Functional Design Creation Tool (FDCT) then uses the structured

المنارة للاستشارات
www.manaraa.com

content and a number of domain specific glossaries from RAT to generate high-level UML class diagrams.

Czarnecki et al. [25] study the relationships between feature models and ontologies. They describe two different approaches, view derivation and view integration, to combine feature models and ontologies using configurable Object Constraint Language constraints.

In [26], [27], Tichy et al. present a modeling tool called the AutoModel and a natural language processing tool called SALE Model Extractor (SALE MX) that aim to improve the requirements engineering process by creating software models from requirement specifications expressed in natural language, using methods such as statistical translation and semantic roles.

AutoModel is consisted of two main components, a UML improver and RESI. RESI is a requirement analyzing tool that can identify ambiguous, faulty, or inaccurate specifications and can offer common sense alternatives.

First, RESI imports a specification in graph format. It then adds information to the specification by tagging part of speech. Using ontology, RESI applies rules to correct common natural language problems, such as ambiguous words, normalization, incompletely specified process words, similar meanings, and nouns without reference index and incorrect usage of universal quantifiers. After the specification is edited, it can be exported back into the original format of the specification.

In support of AutoModel, SaleMX is a suite of natural language processing tools that can automatically generate UML domain models given an annotated requirement specifications expressed in natural language. After the requirement specifications are

المنارة للاستشارات
www.manaraa.com

annotated using a custom made annotation language SALE (SENSE Annotation Language for English), they can be automatically translated into a graph definition for GrGen, a graph rewrite system. The annotated document can then either be translated into an ordinary graph or into a class diagram. Once the UML models are generated, engineers may create executable code using model driven architecture (MDA) tools.

2.2 Design, Implementation, and Integration

After the stakeholders decide on how the end product should behave and have the requirements baselined, engineers begin design the solution. After the architectural design is complete, programmers can then realize the design by coding in the language of their choice. While software productivity has improved significantly with the introduction of object oriented programming (OOP), it is still not enough to satisfy the demand placed on the software industry. Instead of searching for ways of writing code faster, we want to write less of it. As our modern software systems get larger and more complex, the implementation and integration also become increasingly expensive and error-prone. The structure of software system is hidden, or invisible. Software developers, especially the new comers, often need to invest a tremendous amount of time to understand software applications developed in a collaborative environment. Often, the external evidence we have of software is its behavior when executing. It creates the tremendous learning and understanding burden that makes personnel turnover a disaster.

Ontology in design

المنارة للاستشارات
www.manaraa.com

It is the goal of many researchers to automate the process of transforming requirements into design and design into source code. By mapping application domain concepts onto object-oriented concepts, we can then transform knowledge from ontology into programming language, thus generating source code from domain knowledge.

Mapping of ontology into source code

Eberhart describes in [28] two approaches to generate an inference engine and storage repository from RDF schema and RuleML. One is OntoJava. OntoJava is a cross compiler that is responsible for the conversion of RDF schema and RuleML into a set of Java classes. Eberhart also introduced a system called OntoSQL to automatically generate necessary tables/views, and act as a RuleML engine.

Kalyanpur et al. introduced a framework called HarmonIA that automatically map OWL ontologies into Java [29]. It generates a set of Java interfaces and classes, in which an instance of Java class is mapped to a class of ontology with properties, class relationships, and restrictions.

Stevenson and Dobson [30] created a tool called Sapphire that can generates bytecode for a set of Java interfaces corresponding to a set of OWL ontologies.

Design pattern

Design patterns are “descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.” [31] In essence, they are incomplete designs that serve as the templates or the descriptions of solutions to many commonly occurring design problems.

المنارة للاستشارات
www.manaraa.com

Searching for design patterns

Kampffmeyer and Zschaler proposed an approach to help inexperienced developers find design patterns for tasks at hand [32]. The Design Pattern Intent Ontology (DPIO) describes in OWL the relations between design pattern, design problem, and problem concepts, where a *design pattern* is a solution to one or more *design problems* constrained by *problem concept*. By formalizing the intent of design patterns, it is believed that users can retrieve the relevant design patterns based on a rough problem description. Kampffmeyer et al. created the Design Pattern Wizard, a front end interface assisting users in generating well-formed queries.

Component-based software engineering

Component-based software engineering (CBSE) is a relatively new approach to software engineering. CBSE has emerged as one of the most popular paradigms in recent years for its promising results, such as reduced development time and improved productivity. It refers to the software development using reusable parts, also known as components. Components may be a software package, a Web service, or a module that encapsulates a set of related functions (or data).

Code search

But to successfully reuse software components, we must locate them and possibly modify to fit in the software system we are building. Code-search engines provide the backbone for the new generation of reuse support tools. The field of software reuse has

المنارة للاستشارات
www.manaraa.com

come a long way in the past decade, as search engines become more and more sophisticated over the years. Keyword-based searching still remains the most commonly used approach for search engines. It is the easiest to implement, but it is also the most naive approach however. Users must know the exact names of the components to use for each query in order to get relevant results. But many search engines, such as Google Code Search, have gone beyond simple keyword search. In addition to keyword matching, they incorporate other more sophisticated algorithm to retrieve software components. One popular approach is signature matching, which matches a query type with library components. More recently, researchers are trying to apply ontologies to improve the accuracy of the search results.

Searching for code snippets and components

Zygkostiotis et al. introduced a semantic annotation tool that helps engineers search and retrieve software components in [33]. To do so, a software profile is automatically created in the registry and published for every component in a software system. These profiles, known as Advertisement Software Component Profiles, contain inputs from users such as identifier, description, and repository URI, along with information like category, inputs, and outputs that we get by parsing the source code and retrieving the corresponding semantic annotations. During a search, the tool takes the profile attributes as search parameters and matches with the profiles found in registry. Kannan and Srivastava [34] present an approach that extracts the domain knowledge and service abstractions from design diagrams of existing software solutions and represent it in UML format making it more reusable.

المنارة للاستشارات
www.manaraa.com

Happel *et al* [35] proposed an infrastructure for software reuse called KOntoR. KOntoR architecture consists of two major elements: an XML-based metadata repository component to intelligently describe software artifacts independently from a particular format in ontology, and a knowledge component which comprises an ontology infrastructure and reasoning to leverage background knowledge about the artifacts. By combining the explicit and implicit knowledge about the system, KOntoR can derive new information about the system. It uses ontology to integrate information gathered from software artifacts. This resulted central repository of ontologies allows users to efficiently make SPARQL queries to retrieve code fragments needed for an application. Sugumaran and Storey [36] propose an approach to retrieve a reusable design objects from a component repository. They use domain models to stores the objectives, processes, actions, actors, and objects. They combine this information with an ontology that provides information on if/how the objects are related. A prototype was implemented to prove that the use of ontology offers the semantics of the application domain which results in better component searches.

Situation Awareness

Situation awareness [37] refers to "a human appropriately responds to important informational cues. This definition contains four key elements: (1) humans, (2) important informational cues, (3) behavioral cues, and (4) appropriateness of the responses. Important informational cues refer to environmental stimuli that are mentally processed by the human. The appropriateness of the responses implies the comparison of the response with an expected response or a number of possible expected responses."

المنارة للاستشارات
www.manaraa.com

In software engineering, researchers study situation-awareness to handle information overload situations in an enterprise scale software systems. We design situation-aware systems as a preventive measure.

Matheus et al. [38] describe a situation-awareness application called SAWA that uses SWRL / OWL to represent a supply logistics scenario. They also

Situation-awareness is not a trivial task. Situations are often hard to predict. Currently, SA techniques are only able to detect situations that have already occurred in the past, and, hence, are not applicable for predicting critical situations before they occur for the first time. However, critical situations that endanger life usually do not occur frequent enough to obtain meaningful training data for machine learning.

Baumgartner et al. [39] propose using different ontologies to describe qualitative facts for achieving situation awareness, along with techniques thereupon predicting future situations without historic data.

2.3 Testing and Verification

After a software system is implemented, the next step is get the software tested to ensure quality. Software testing, also known as verification, is an important phase in the development cycle. It is the process of validating and verifying that the product meets the requirements set forth by the customers and that it functions without any defects or bugs. Software testing can generally be categorized into black box testing and white box testing:

المنارة للاستشارات
www.manaraa.com

Black box testing is when the tester has no knowledge of the internal workings of the product. White box testing, on the other hand, refers to the method. Unit-testing is an example of white box testing.

Software testing can be very labor intensive and costly. For many years, researchers look for ways to automate the software testing process. There are generally two parts to test automation. First, we need to generate test cases. Then, we execute them. Commonly, many engineers write scripts to auto execute tests on software systems. This type of automation still requires manual generation of test cases. Ideally, we would like to auto generate and execute test cases, minimizing or eliminating all human intervention.

Ontology in Verification

Ontology can help ensure or improve software quality in a number of ways. To better manage software testing, ontologies have been created by engineers to describe various testing tasks during the test process. As software systems become larger and more distributed, agent-based testing becomes a popular approach. Ontology may be used to facilitate the communication and interaction between agents.

Code Review

A number of researchers have also proposed to use ontology to create a query service that can help detect bad code smell. Code smell is a warning sign of a potential problem in software design. Unlike errors, code smell does not necessarily result in

المنارة للاستشارات
www.manaraa.com

errors. It is simply an indication of a poor design that may eventually lead to a deeper problem. To get rid of the bad code smell, source code often gets refactored.

In [40], Luo et al. present an ontology that describes the relationship between code smell and anti-patterns. An anti-pattern is essentially a commonly recurring design solution. They formally describe the concepts of anti-patterns, code smells, and the corresponding refactoring solutions in a knowledge domain model. A formal representation of this interrelationships provides guidance to developers in the removal of code smell. By eliminating anti-patterns in the source code, we believe that the quality of the software will improve and be less error-prone. It is also believed that the source code automatically becomes easier to read and understand.

In [41], Yu et al. describe bug patterns in SWRL rules and Java program specification in an ontology model. They propose parsing Java source code into the Abstract Syntax Tree (AST), then automatically map it to the program ontology model. The two models are then exported into an OWL file, which serves as an input to a rule engine. The output of the rule engine should output inference results, indicating whether there is a bug.

Annotations and Testing

One of the most labor intensive types of software testing is GUI testing. The reason being that it is not trivial to include all the possible sequences of event interactions. A few researchers propose using annotations to better understand the possible inputs and outputs.

www.manaraa.com

Annotation is a note added to a text. In software engineering, programmers often add annotations to the source code to improve readability and explain design rationale.

In [42], Rauf et al. proposed combining the use of ontology and semantic annotation to automatically generate test cases for GUI testing oracle development. In this approach, the ontology is built using document specifications, expert opinions, the interactions in the GUI test framework, as well as the event-flow model that represents all possible sequences of events that can be executed on the GUI. Semantic annotations for test cases can be derived from this ontology.

Auto-generate test cases and test case reuse

Test case creation is not a trivial task as it requires some domain knowledge and the complete understanding for the requirements. A number of researches have demonstrated success in using ontology to represent the required knowledge needed for test case generation.

In [43], Wang et al. proposed using semantic models to generate test data. Models, such as OWL-S, can provide both behavioral information and semantic information on the datatypes. They use the Petri-Net model to describe the structure and behavior of a composite service. Together with the ontology in OWL-S, test cases can automatically be generated.

Nasser created a knowledge based framework to generate unit tests from UML state machines [44]. It exports the test criteria and high-level test descriptions in the form of an ontology and rules. The ontology also contains information about the generated test suites, allowing redundancy checking using reasoning.

للاستشارات
www.manaraa.com

Test automation

In an ideal world, activities in the testing phase of the waterfall model would be completely automated without any human intervention. This means we need to automate test case creation and execution. In order to automate such process, we need to define the expected input and the expected output in a machine processable format.

Ontology as test oracle

Test oracle is "a mechanism used by software testers and software engineers for determining whether a test has passed or failed. [64]" Traditionally, software testers would act as the oracle. In recent years, a number of research activities have been done to automate the test oracle, in hopes to speed up the test process and reduce the overall cost of software development.

Ontology can be used to represent domain knowledge in a machine processable fashion, making it possible for agents to communicate with each other.

Maamri and Sahnoun [45] propose using agents to automate the generation and the execution of test cases. This multi-agent system, known as MAEST, consists of four types of agents: administrator which manages the entire system, testing which supervises the entire testing process, interface which simulates the interaction between the system and the users, and helping agents which perform various tasks such as test case generation and act as the test oracle and determine the verdict of every test. Each of the agents must understand the test process and be able to communicate with one

المنارة للاستشارات
www.manaraa.com

another. To do so, a software testing ontology is designed to include test information such as activity, method, capability, and artifact.

Nguyen et al. [46] developed a testing framework called eCat that not only supports the (semi) automatic generation but also the execution of tests cases used for multi-agent systems (MAS) testing. It consists of three main components: the Test Suite Editor takes test data represented as Tropos-supported goal analysis diagrams and semi-automatically test suites; the Autonomous Tester Agent is responsible for executing test suites on a MAS; and the Monitoring Agent monitors communications among agent for debugging purposes.

Test Documents

Serhatli and Alpaslan present in [47] an ontology-based automated question answering system on software test document domain. They propose an algorithm to transform queries in free text into an expression to be interpreted by a DL reasoner. To do so, test documents are first described in a common language, OWL-DL. Then they use the web interface to send queries to the Pellet Server. Question types are limited to who, what, when, which, and how many.

2.4 Maintenance

Software maintenance is “the process of modifying a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment.” [48] Managing and maintaining software systems require a tremendous amount of knowledge. Not only do we need to know the problems we need

المنارة للاستشارات
www.manaraa.com

to solve and all its limitations / restrictions, we also need to concern ourselves with the process used, language used, the architecture, how the parts fit together, as well as how the system interacts with its environment. It is usually costly and time-consuming to gather this knowledge. Therefore, it is important to find a way to efficiently store this information.

Modeling software maintenance processes

One use of ontology in software maintenance is to model its processes. Formalizing the relationship between the maintenance-related objectives and the maintenance activities can help us quickly identify the relevant resources needed to reach the goal.

One of the earliest works to model software maintenance process in the form of ontology was done by Kitchenham et al. Kitchenham et al. proposed in [49] to an ontology that defines the investigation activity, modification activity, management activity, quality assurance activity, and resource.

Ruiz et al develop an ontology called MANTIS [50] that extends Kitchenham's work. MANTIS is made up of three ontologies. The static aspect is defined in the maintenance ontology, and the dynamic aspect is covered in the workflow ontology. A measurement ontology is also introduced to represent both static and dynamic that are not already capture by the other two ontologies.

On the other hand, Dias et al [51] create an ontology that focuses on the knowledge about the software system itself, the maintainer's skills, the maintenance activity, the organizational structure, and the application domain.

المنارة للاستشارات
www.manaraa.com

Program Comprehension

Program comprehension, or reverse engineering, is a research area that aims to develop tools and methodologies to improve the understanding of software systems. It focuses on recovering design and other relevant information from a system. Before a software application can evolve in a controlled manner, engineers must first understand the legacy or existing software at hand. Therefore, it is an essential though often costly activity in software maintenance.

Meng et al. presented in [52] a formal process model to support program comprehension using ontology and description logic. It includes information and the interrelations of task, user, tool, artifact, software artifact, documents, and historical data. Abebe and Tonella [53] described an approach to support program understanding by extracting concepts and relations from the source code. Given the source code as input, a term list is first created by decomposing the element names using camel case.

Configuration Management

As software evolves, changes are inevitable. With today's large and complex software systems, they may become a disaster if not handled correctly. Software configuration management is [54] "unique identification, controlled storage, change control, and status reporting of selected intermediate work products, product components, and products during the life of a system."

Shahri et al. [55] propose an approach to formalize and integrate the local configuration constraints and version restrictions using OWL-DL ontologies. Ontology can help us organize information needed for various activities in configuration management. Not

المنارة للاستشارات
www.manaraa.com

only does it help in planning process, it also helps to identify thousands of software artifacts that may be associated with each software release. In addition, ontology allows us to query relevant information specific to a release and infer new facts. Furthermore, software building tools may ontology to identify dependencies that must hold between versions.

Modeling Software Artifacts

Software artifacts are essentially any items that are developed and used during the software development cycle. They may include any test cases, requirements documents, and design models. Ontology helps engineers create mappings between software artifacts, making sure that they remain consistent as the software evolves. It also allows engineers to query the changes made to a software application.

Hyland-Wood et al. [56] describe relationships between all object-oriented components, also known as the Software Engineering Concepts (SECs) in the form of ontology. The ontology models the Java language structure and represents information such as classes and methods, tests, metrics, and requirements.

Tappolet et al. presented an extension to FAMIX with a set of OWL software ontology models called EvoOnt [57]. EvoOnt includes three different models, namely the software ontology model (SOM), the bug ontology model (BOM), and the version ontology model (VOM). They include information such as software design, release history, and bug reports. Tappolet et al also developed a semantic-based framework that is based on the SPARQL query language. Today with EvoOnt, iSPARQL is an extension of SPARQL that may be in software analysis.

المنارة للاستشارات
www.manaraa.com

Witte et al. [59] created a uniform ontological model for source code and software artifacts. They created a software ontology that consists of two sub-ontologies to represent the source code and its corresponding documents. The source code sub-ontology contains entities, such as methods and variables, and their associated actions. The documentation sub-ontology covers concepts such as programming languages, algorithms, data structures, and design decisions. Representing software artifacts in the form of ontology allow engineers to share common concepts between source code artifacts and documentations. It helps to establish traceability links between the different resources, making the integration of information easier. Not only can ontological models help us in program comprehension, techniques such as DL and other reasoners can check for consistency of the artifacts.

Documentation

Software documentations refer to any written texts associated with a software project. This may include requirements specifications, design documents, APIs, and user manuals. The quality of these documentations is important in program comprehension and for software reuse. As software systems get larger and more complex, it becomes increasingly difficult and costly to maintain and manage these documentations. Ambrosio et al.[60] propose an ontology-based approach to generate up-to-date and consistent software documentation. They believe the key to create good documentations, we must first find a way to organize and manage the artifacts generated by the software development process. To do so, they suggest storing such data in two ontologies. A structural component ontology may be used to store the

www.manaraa.com

different types of artifacts based on their internal structure. A domain component ontology, on the other hand, may be used to represent the semantic relationships between the artifacts described in the previously mentioned ontology.

Witte et al. also presented a tool called JavadocMiner [61] that can automatically assess the quality of source code comments and export the result to an ontology. The tool performs a set of heuristics to evaluate the quality of the comments expressed in natural language and to check for the code/comment consistency. Together with an OWL reasoner, JavadocMiner outputs an ontology that models the NLP related entities, source code identifiers, comments, and entities in comments.

Security Ontology

Fenz et al. [62] study security in small and medium size enterprises (SMEs) and propose a holistic solution based on a security ontology that includes low-cost risk management and threat analysis. The security ontology consists of five sub-ontologies. “Threat” is the main sub-ontology and includes proper countermeasures, threatened infrastructures and proper evaluation methods. “Attribute” sub-ontology models the impact of threats, “Infrastructure” describes infrastructure elements, “Role” maps enterprise hierarchies and “Person” represents natural persons who are relevant for security issues modeling.

Undercoffer et al. [63] analyzed around 4000 vulnerabilities and their exploit strategies and after that they created an ontology, in DAML+OIL and DAMLJessKB, for specifying computer attacks. They also summarize the main languages for specifying computer attacks, including P-Best, STATL, LogWeaver, CISL, BRO, Snort Rules and IDMEF.

المنارة للاستشارات
www.manaraa.com

Then they present several use case scenarios with common attacks, such as “Denial of Service – Syn Flood”, “The Classic Mitnick Type Attack” and “Buffer Overflow Attack”.

المنارة للاستشارات
www.manaraa.com

CHAPTER 3

In the previous chapter, I gave an overview of the current research activities in semantic software engineering. In this chapter, I will present an application that incorporates the semantic technologies to improve annotations added by software developers. Using this semantic annotation tool, engineers should easily be able to answer questions such as:

- What does this method/function do?
- Can I reuse this class?
- Where is this method/function being called?
- Will the code still behave the same if I remove this line?
- Why is this design approach used?
- Where is this variable being referenced?

Motivation

In order for a software system to be evolved and maintained, developers must first understand how the existing code works and why a certain design approach is made. Many programmers often underestimate the importance of adequate inline documentations because they assume their code is straightforward and easy enough for others to understand. Of course, we know that is usually not the case, as the lack of good documentations often makes personnel turnover a disaster.

For software developers, good documentations not only give us a brief description of the software component, but also tell (or remind) us how it may be (re)used and perhaps reveal the design rationale. This way, the author or the programmers in the

المنارة للاستشارات
www.manaraa.com

same project will not need to read every line of code trying to figure out what the code does and / or why a certain approach is used years from code complete. For software testers, good documentations reduce ambiguities and confusion over the expected behavior of a software system when white box testing is being performed. Nonetheless, documenting software is indeed a tedious task. But while it is a time-consuming activity, in a long run good documentation may not only help developers write better and less error-prone code, but also improve overall productivity.

So what defines good documentations? And how may an annotation tool help? Just like many other development tools, the ultimate goal of an annotation tool is to help programmers write good code faster. It provides a controlled vocabulary to describe every software entity in question. A predefined list of terms allows for no ambiguities in meaning, thus making it possible for machines to provide better answers to queries on the existing code.

Semantic Annotations for Software Engineering

As mentioned earlier in this thesis, annotations are often added to software artifacts to improve readability and explain design rationale. An annotation is, by definition, a critical or explanatory note. In software engineering, annotations are meant to provide information about a program that is not part of the code itself. Sometimes, annotations and comments in code are used interchangeably. Conventionally, programmers use annotations to:

- Planning -- Programmers sometimes put pseudo code in the comments. They may also explain the logic behind the code in the comments.

www.manaraa.com

- Code description -- Programmers often clarify their intent in the comments. They may put any constructive criticism in there to remind one that the code needs rework in the future
- Algorithm description -- Programmers may explain the rationale behind why a certain approach is chosen. Programmers may explain the algorithm in greater details if the novel approach is taken.

In Java, annotations are structured syntaxes, similar to the programming language. Not only may annotations be used for documentation purposes, compilers may also use the annotations to detect errors or suppress warnings. They may also used to generate auxiliary source code. However, that is beyond the scope of this thesis.

Generally speaking, there are no restrictions to what and when to annotate. But the best annotations should help the programmers, new or experienced, understand the code and prevent us from reintroducing old bugs. Ideally, they may provide details on how to potentially reuse existing code fragments. Even if we are unable to reuse the existing code fragment, we may still be inspired by it.

In the world of semantic computing, annotating is sometimes known as tagging. In the recent years, tagging is a popular way to assign a keyword to a file, whether it is a digital image or a blog post. Doing so helps in identification / classification and makes searching easier. But while tagging is a great way to classify an item, a tagging system does not have any information about the semantics of each tag.

Semantic annotations are not quite the same as tags. We can think of semantic annotations as tags, enriched with meanings. They include information on how entities

للاستشارات المنارة
www.manaraa.com

are related and associated with one another, allowing queries to retrieve more data, including information that may not be explicitly related to the original search.

An annotation tool comes in handy in this regard. It helps to capture syntactic and semantic descriptions of a software component and its functionality.

A Tool for Semantic Software Engineering

The purpose of an annotation tool is twofold. It should (1) guide users in the annotation process so vocabulary may be controlled, and (2) create semantic annotations for retrieval and better query answers in the future. To do so, the annotations need to be converted to a machine processable format. I chose to translate this domain knowledge into Web Ontology Language, for it has a larger vocabulary and stronger syntax than RDF. More importantly, it provides better reasoning and inferences, given a reasoner. My tool guides its users through a list of basic questions about the code, and stores the answers in a form of ontology for future use. These questions include, but not limited to:

- What is the expected behavior of this piece of code fragment?
- What does this variable store?
- Where is this method/function referenced?

They also serve as the competency questions for the ontology. To answer the above questions, I developed an ontology that describes both the expected behavior and the low level design. Users may define the expected behavior in both natural language (free text) and formal language (unit test cases). Developers can learn the general

المنارة للاستشارات
www.manaraa.com

functionality of a software component and get a basic understanding of the API by looking at the unit tests. Unit test cases can oftentimes act as a dynamic documentation. However, it is expensive and impractical to create an exhaustive test suite for every component in a software system. This is the reason why I believe that it is best to combine documentations expressed in both natural language and formal language.

Modeling Source Code

An ontology to model source code written in object-oriented programming languages, such as Java or C++. Even though object-oriented programming languages all differ in syntax and in semantics, most of them share the following same fundamental concepts:

- Classes of objects
  - A class describes an object with a state and a behavior
- Instances of classes
  - An instance is an occurrence of an object. Every instance of a class has the same set of attributes, but the value of the attributes vary from instance to instance
- Methods (also known as function or procedure)
  - A method is essentially a subprogram that acts on data, and may or not may not return a value. It defines a behavioral property of a class
- Subtype polymorphism
  - A type is a subtype if it supports every property of its supertype.
- Inheritance

المنارة للاستشارات
www.manaraa.com

- ➢ A type inherits from its superclass if it only has a subset of its parent's properties.

The meta-model I use to model semantic annotations is based on FAMIX core, a programming language-independent model for representing object-oriented source code.

The model contains many fundamental object-oriented concepts described above, where a code fragment is an entity. Beneath Entity, at the top level, are the following subclasses and properties:

Subclass – SourceLanguage

- Subclass -- Java
    - Disjoints with Python, Cpp, and OtherLanguage
- Subclass -- Python
    - Disjoints with OtherLanguage, CppLanguage, and Java
- Subclass -- Cpp
    - Disjoints with PythonLanguage, OtherLanguage, and Java
- Subclass -- OtherLanguage
    - Disjoints with Python, CppLanguage, and Java

Subclass -- SourceAnchor

- represents the pointer that references the location of the code fragment. It should tell us where to find the code fragment
- Subclass – FileAnchor
    - Disjoints with TextAnchor
- Subclass – TextAnchor

المنارة للاستشارات
www.manaraa.com

- Disjoints with FileAnchor

Subclass --- Type

- represents either an abstract class, an interface, or a concrete class.
- Subclass – AbstractClass
  - Disjoints with ConcreteClass and Interface
- Subclass -- Interface
  - Disjoints with ConcreteClass and AbstractClass
- Subclass -- ConcreteClass
  - The domain is Class, and the range is Class.
  - Disjoints with PrimitiveType
- Object property -- hasSubclass / isSubclassOf
  - The domain is Class, and the range is Class
- Object property -- hasMethod / isMethodOf
  - The domain is Class, and the range is BehaviorEntity
- Object property -- hasSubtype / isSubtypeOf
  - The domain is Class, and the range is Class.
- Object property -- hasAccessModifier / isAccessModifierOf
  - The domain is Class, and the range of AccessModifier.
- Object property -- hasAttribute / isAttributeOf
  - A class may have a handful of data property.
  - The domain is Class, and the range is Attribute.

Subclass -- StructuralEntity

المنارة للاستشارات
www.manaraa.com

- Refers to a data container with a static symbolic name but a dynamic value. It has the following subclasses.
- Disjoints with BehaviorEntity and Type
- Object property -- References / isReferencedBy
- Subclass -- Attribute
- Subclass -- LocalVariable
  - A variable given local scope.
  - Disjoints with GlobalVariable
- Subclass -- GlobalVariable
  - A variable that is accessible in every scope
  - Disjoints with LocalVariable

Subclass -- BehaviorialEntity

- Code that performs a task
- Disjoints with StructuralEntity and Type
- Subclass -- Function
  - Disjoints with Method
- Subclass -- Method
  - Disjoints with Function
- Object property -- hasLocalVariable / isLocalVariableOf
  - A variable accessible only within the method. The domain is BehaviorialEntity, and the range is LocalVariable.
- Object property -- calls / isCalledBy
  - A function / method may call or be called by another function / method.

المنارة للاستشارات
www.manaraa.com

    - The domain is BehavioralEntity, and the range is BehavioralEntity
- Object property -- hasExpectedBehavior / isExpectedBehaviorOf

Subclass -- ExpectedBehavior

- Describes the expected behavior of a method or a function
- Data property -- Input
    - If the code fragment is a method or a function, this is where the user enters the input(s) or the parameter(s).
- Data property -- Output
    - If the code fragment is a method or a function, this is where the user enters the expected output if there is one.
- Data property -- Detail
    - Detail may be any free-text that reveals more information about the expected behavior of an entity. It may be design rationale, change / update information
- Data property -- TestSuite
    - In additional to the natural language description, users can express the expected behavior using test cases.

Subclass -- AccessModifier

- Internal
    - Disjoints with public, protected, and private
- Public
    - Disjoints with private, internal and protected

المنارة للاستشارات
www.manaraa.com

- Private
    - Disjoints with public, internal and protected
- Protected
    - Disjoints with protected, internal, and public

Implementation

To prove the concept in the previous section, I created a prototype based on the approach described above. This annotation tool is implemented in Python. It is essentially an interactive word editor written using wxPython [65], the Python version of the GUI API wxWidgets. The tool allows users to highlight a section of the source code, and will guide users to enter relevant information about the source code.

Expressing annotation in OWL

Once the tool captures all the data needed, it stores all the relevant information in the form of an ontology using FuXi [58], a Python library for all things semantic web related. One of the primary modules in FuXi is Syntax. FuXi.Syntax incorporates the InfixOwl library, which is based on the Manchester OWL syntax. It is the Python binding for OWL Abstract Syntax that allows us to create or modify an ontology in OWL.

Retrieving annotations

المنارة للاستشارات
www.manaraa.com

The InfixOwl library is capable of parsing OWL/XML files, thus also allows the tool user to highlight a code fragment and retrieve information described in an existing relevant ontology.

Running XUnit test cases

Upon request or changes, the tool retrieves and runs the relevant xUnit test cases to ensure that the code behaves as expected. Xunit [66] refers any code driven unit testing frameworks that are patterned on JUnit (Java) or SUnit (SmallTalk). SUnit and JUnit were originally implemented by Kent Beck. Since then, the framework has been ported to a number of other programming languages, including C++ (CppUnit) and Python (PyUnit). They are all the de facto standard unit testing framework for their respective languages. They let users write repeatable tests to ensure its functionalities after refactoring or a bug fix. Tests can be aggregated into a test suite so they can all be run in a single operation. The test results can either be outputted on the screen, or saved as a text file.

Example - Annotating quick sort

To demonstrate the practicality of the tool, let us try using it to annotate a simple program, quick sort. I implemented this algorithm in Python (Appendix A). Quick sort is a classic sorting algorithm. In order to perform quick sort, we need to:

1. Choose a pivot, which may be any number from the input array.

المنارة للاستشارات
www.manaraa.com

2. Partition the array into two empty sub-arrays. Rearrange the inputs so that the numbers smaller than the pivot move to the left array and the numbers larger than the pivot go to the right array.

3. Recursively sort the sub-arrays

Highlighting code fragment

Users can highlight any section of the source code. It may be a class, a method/function, or simply a variable. Using the file location and the indexes of the highlighted code, the tool can determine the sourceAnchor of the code fragment. Then based on the file extension of the code, the tool will also know what programming language is used. Languages supported include Java (.java), Python (.py), and C++ (.cpp).

Storing domain knowledge

After the user is finished annotating, the tool will store this information in OWL/XML in the same directory as the source code. The tool also increments the version number whenever user tries to save changes made to the source code. (See Figure 3.1)

Retrieving domain knowledge

If the source code has previously been annotated, the tool will display the existing annotation for the highlighted code. It will then ask the user if he or she would like to add new additional annotation. (See Figure 3.2)

المنارة للاستشارات
www.manaraa.com

Running test cases

To ensure new changes do not introduce bugs, the tool can run the defined PyUnit test cases (Appendix B) upon the code updates or request by users. (See Figure 3.2)

Querying semantic annotations

Unlike RDF, OWL does not yet have a standard query mechanism like SPARQL. But it does not mean searching is not possible. Researchers have proposed a number of query languages, such as SQWRL and SPARQL-DL. SPARQL-DL is a quite expressive language which particularly allows to mix TBox, RBox, and ABox queries.

Example:

Type (?y, Method), PropertyValue(?y, hasMethod, toString)

On the other hand, SQWRL (Semantic Query-Enhanced Web Rule Language) is a SWRL-based query language that can be used to query OWL ontologies. SQWRL provides SQL-like operations to format knowledge retrieved from an OWL ontology.

Example:

Interface(?i) → sqwrl:select(?i)

There are Protégé plug ins for both query languages.

Reasoning with an Inference Engine

المنارة للاستشارات
www.manaraa.com

FuXi comes with two top-down (backward chaining) algorithms for SPARQL RIF-Core and OWL 2 RL entailment.   We may also the Fuxi command line script to has support for efficient backwards and forward chaining to solve the answers to a user-specified query.

Alternatively, we may use Pellet for OWL reasoning. There are multiple interfaces to the reasoner, including a command line program and a programmatic API allowing a standalone application to access its reasoning capabilities. It may also be integrated with Protégé, or used concurrently with the popular Semantic Web framework Jena.

المنارة للاستشارات
www.manaraa.com

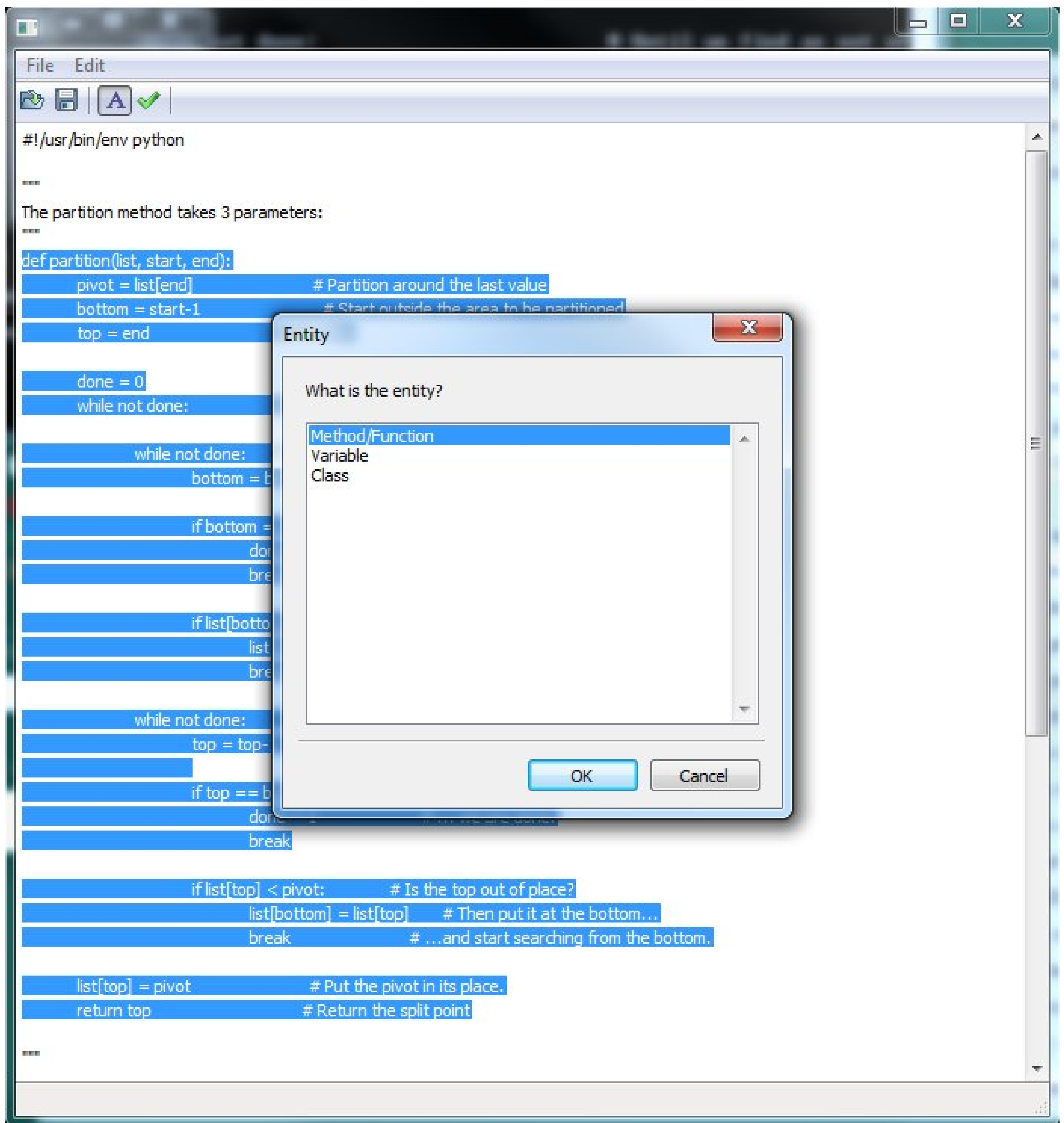

Figure 3.1

المنارة للاستشارات
www.manaraa.com

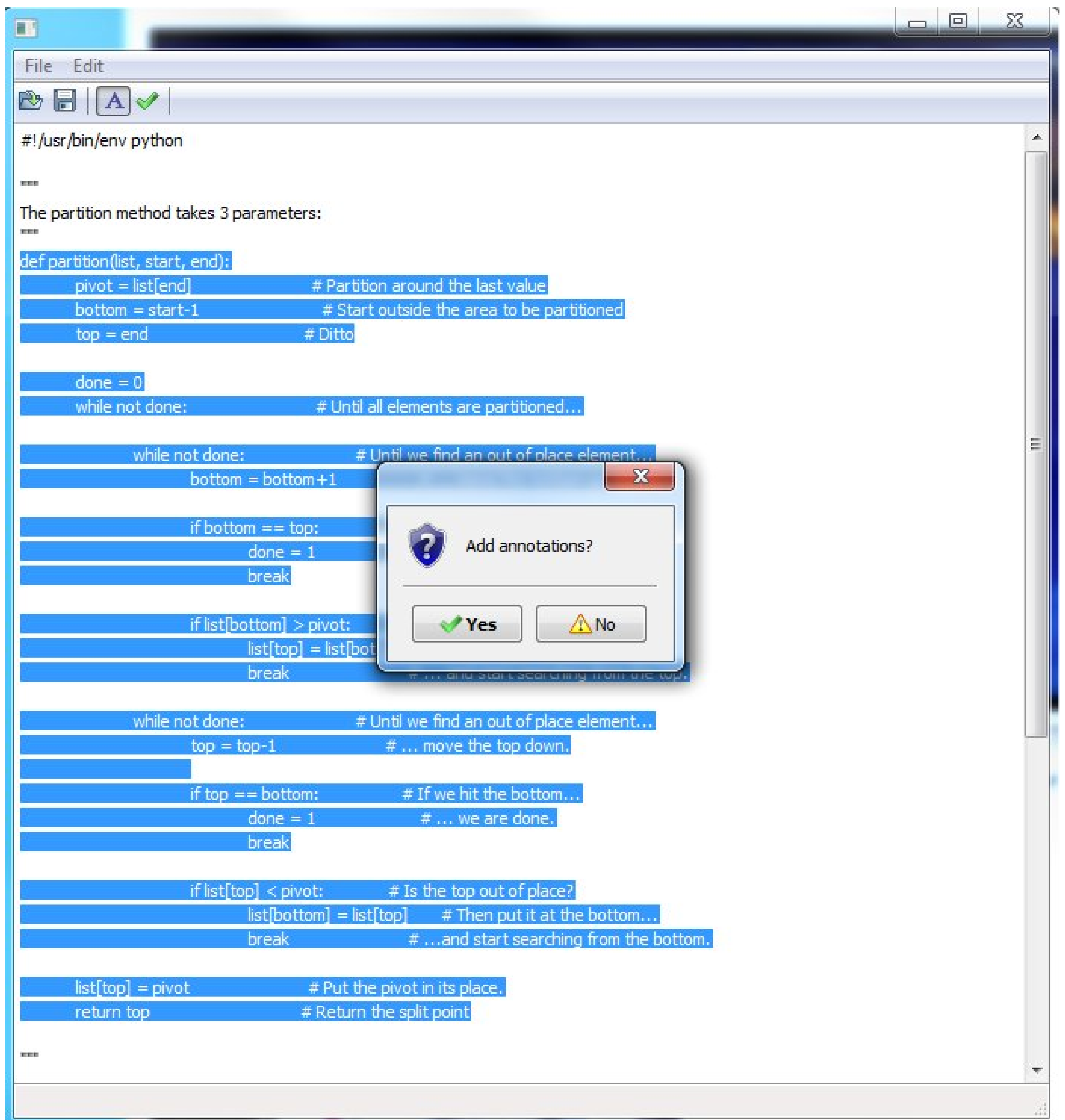

Figure 3.2

المنارة للاستشارات
www.manaraa.com

Benefits and Other Potential Uses

As discussed previously, there are many benefits to modeling source code in ontology. One main use is for program comprehension. Engineers may also use this as a tool to detect code smell in the code and encourage best practice design solutions.

One of the commonly seen anti-patterns in object oriented programming is God class (also known as a blob class). A God class refers to an object that does everything. It is bad practice because it merges data and process, making the class process oriented (also known as the procedural design). A God class makes a software component too complex for reuse and testing. A good way to find such a class is to look for a class with over 50 attributes and behavioral entities.

Another anti-pattern that may be detected is object orgy. It refers to objects that are insufficiently encapsulated. Object orgy is most commonly seen in Perl. However, it may happen in other programs written in any object-oriented languages that support the use of access modifiers. When there is an object orgy, objects have access to another objects' attributes and may result in indiscriminate passing of objects. To detect object orgy, we may look at all the public objects and see how they are being referenced and called.

We can also use the ontology to determine if we have a Swiss army knife (also known as a Kitchen Sink). In a Swiss army knife situation, there are more interfaces than necessary. An easy search will tell us if there are more interfaces than implementation classes.

In addition, we can look up the number of constants and where it is being referenced and initialized to determine if we may have constant interface problem in a large

المنارة للاستشارات
www.manaraa.com

software system. Constant interface happens when constants are defined in the interface. It is considered a poor design because constants are often considered an implementation detail and do not belong in the interface.

But how is this model different than the other models?

First and foremost, this model is language independent. Concepts in this model are universal fundamental concepts commonly found in most object-oriented programming languages. This is especially useful for developers working on a software application that uses multiple programming languages.

Secondly, this tool allows users to annotate source code in both free-text and formal language.   Mapping test cases to the relevant code helps us check for consistency between the expected behavior specified in free text by the original author of the code.

Limitations and Assumptions

This tool is a work-in-progress prototype. There is still a lot of room for improvement. Known limitations are as follows:

Unit Tests

Currently, the tool only supports xUnit test frameworks. I also assume that the users of the tool already know how to write xUnit test cases.

Manual annotation

This tool relies on the users to manually enter information about the source code. There is also no checking to ensure that the annotations remain consistent with the behavior of the software component after any code changes.

المنارة للاستشارات
www.manaraa.com

Viewing

Currently, users can only view the retrieved annotations in plain text, or rely an ontology editor to view the interrelations between the defined entities graphically.

Searching

Users of this tool are expected to be familiar with an OWL query language, because no guidance is currently provided. This may create a learning curve for those who are not already fluent in either SPARQL-DL or SQWRL.

Discussion

Semantic software engineering is the research area that studies the technologies used to derive semantics from software artifacts, such as source code, requirements specification, and documentations. These semantic technologies not only give us a means to formalize domain knowledge, but also enable tools to improve quality and reusability of software, along with the communication among its stakeholders. Nonetheless, software semantics do not come cheap. Ontology creation, for starters, is a tedious task.

One of the biggest challenges faced when I was creating the tool was determining how to store the annotations. While there is no “correct” way to model a domain, the ontology should at the minimum be able to answer the competency questions. Furthermore, creating an ontology that is language independent means that I need to have the basic knowledge in a number of programming paradigms. It is definitely not a trivial task.

Because developing an ontology is such a time-consuming task, I learned that it is

www.manaraa.com

generally a good idea to first check to see if there exists a reusable ontology that we may modify or extend. I went through a number of ontologies that I may reuse to describe source code. At the end, I picked FAMIX for its complete coverage of the concepts in many object-oriented programming languages.

Moreover, it was not a straightforward task to capture every single concept shared by every single OOP language. For example, some may argue that Java is not truly object-oriented. I studied a number of programming languages that are arguably the more popular OOP languages, including Java, C++, and Python.

Another challenge I faced was the lack of support in semantic technologies for Python. Most Semantic Web software is built with Java. There is close to no OWL support for Python. As far as I know, there is only FuXi that is currently still under active development.

With that said, there are more and more supporting tools for OWL. For example, Oracle Database 11g supports the storage of semantic data and ontologies. We can also perform ontology-assisted queries on relational data, and use either the built-in or a third party reasoner when querying on semantic data.

Future Work

The roadmap for the annotation tool includes:

User feedback

The objective of an annotation tool is to facilitate software reuse and to lower the learning curve for new programmers. Therefore, it is important to get feedback from users to see if the tool helps them or not. Does the tool encourage engineers to

المنارة للاستشارات
www.manaraa.com

annotate source code? Does the tool help new programmers understand the software better and faster?

Versioning details

In some software versioning and source revision systems, such as Subversion, developers are asked to annotate the updates they commit to the source code. By expanding the ontology to include such annotation, other programmers can easily look up this change information and know who to ask for more information if need be.

Tracking bug fixes and feature updates

Along with the annotation stored in the source revision systems, bug tracking and feature request systems such as Bugzilla and JIRA may also provide a lot of insights to programmers. Conventionally, the programmers include a referenced bug report number and a brief description of the changes made. We may find details on past and current outstanding issues found in a software system. By understanding the bugs found in the past, new programmers may be able to avoid making the same mistakes in the future. This information may also provide insights on what changes may have caused new bugs.

Support more language specific concepts

In the future, the ontology used by the tool may be extended to support language specific concepts. For example, we may include access modifiers. They affect how a method may be reused, even though the same concept does not apply to other programming languages such as Python.

Support more unit test frameworks

Since the ultimate goal is to create an annotation tool that supports as many

المنارة للاستشارات
www.manaraa.com

programming languages as possible, it would make sense to include more than one test framework, allowing automated test execution for every language. In the future, we may expand the list of supported unit testing frameworks.

Automatic query creation

With information expressed in OWL, we can take advantage of the reasoning capability and its powerful search functionality using query languages, such as SQWRL. However, one needs to learn the query syntax. Therefore, I would like to create a user interface to guide users in creating such queries.

Reasoning Support

One of the main advantages of representing domain knowledge in OWL is its reasoning capability. It provides logical inference, given the information at hand, thus providing better solutions for commonly asked questions by programmers. As of today, there exist no reasoners written in Python. However, we can use PelletServer as the backend reasoning system. It appears to be universally accessible via a REST interface, allowing a Python program to get its service by issuing an HTTP GET.

Adding Rules

While OWL allows us to adequately model software components and service by defining terms to describe the hierarchical structures and their properties, the expressivity of OWL is not sufficient to describe implication rules, modalities and probabilities that are needed for a proper reasoning in determining whether a component may be reused. In the future, I would like to add rules to support behavior variability in response to different situations in various environments.

المنارة للاستشارات
www.manaraa.com

Semi-auto / Auto annotation

Currently, this tool requires manual annotations. While the benefits of storing annotations in a form of ontology outnumber its drawbacks, the annotation process may be too tedious for many engineers and defeat the purpose of the tool which is to speed up the development process. Therefore, in the future, I would like to look into automating parts of annotation process, freeing users from having to manually identify code fragment. Parts that may be good candidates for automation include the structure of the code fragment. We may identify the type of code it is by learning the reserved keywords with specific meaning and predefined functions. We may also get some insights as to what a function/method is expected to do by dissecting its identifier.

المنارة للاستشارات
www.manaraa.com

Figure 3.3

المنارة للاستشارات
www.manaraa.com

Chapter 4

This chapter will conclude this thesis with a discussion of what is still missing and a proposal of future research ideas for the field of semantic software engineering. I will also wrap up by summarizing this thesis at the end.

4.1 Future Research Directions

Although we can see that semantic software engineering has come a long way [68], more effort is still needed before machines can perform the entire development cycle without human interventions.

Requirements engineering framework

Requirement engineering has received tremendous amount of attention in the semantic computing community. Researchers have approached a number of approaches to use ontologies to detect inconsistencies, ambiguities, and incompleteness in requirements. Researchers should focus on creating a framework that allows requirements engineers to not only describe requirements specification documents, but also to formally represent requirements and the application domain knowledge. Having an integrated framework allows stakeholders to collaborate and significantly improve RE processes.

Automating testing

While engineers can now easily automate the execution of unit tests, we must still first manually develop the test cases. Test case generation is an active research area with lots of potentials. Many researchers have proposed creating test cases from API or formal specifications. In the future, researchers may consider generating automatically

المنارة للاستشارات
www.manaraa.com

from requirements in natural language. Other potential applications of ontologies in software testing include test planning and test oracle. Ontologies may be used to describe activities and tasks commonly performed in a test cycle. The automated reasoning capability also helps automate test oracle. They may also focus on creating a test framework that can automate the entire test process, from test case generations to verifications.

Automatic documentation generation

It is often more difficult to analyze source code with in-line documentations because we need to first differentiate free text from the formal language before applying NLP techniques on the comments in the source code . There are currently a number of tools that use the special tags or other semi-structured language embedded in the source code to automatically generate APIs (Java) or documentations in XML format (C#). In the future, researchers may try to generate documentation by extracting semantic information directly from the source code, to avoid inconsistency between the code and its relevant documentation.

Quality assessment

Currently, there are a number of approaches proposed to use ontology to evaluate the quality of requirements specifications. A quality ontology may be used to represent metrics for software evaluations. Together with other ontologies that describes the source code and other software artifacts, it may be used to assess the quality of software artifacts.

Troubleshooting and understanding proprietary systems

Today, research efforts tend to focus on open-source software systems due to its

المنارة للاستشارات
www.manaraa.com

availability. Engineers and software users can, however, benefit greatly from a repository of useful troubleshooting tips for proprietary systems. Ontology may be used store and organize such knowledge gathered from forums and users documentations.

Code smell / Bug detection

Often times, it takes longer to fix a bug than to implement a new feature for a software system. Ontology may be used to store the ever-growing bug information for future reference and for sharing with other developers. In the future, researchers can incorporate knowledge described in a bug ontology, a design pattern and an anti-pattern ontology in a tool to help detect bugs and suggest ways to reverse the code smell in software systems.

Transforming requirements into software artifacts

Ultimately, we would like to simply be able to transform requirements expressed in natural language by customers into source code and other software artifacts. While many IDE's can already automate daily mundane tasks for developers, there is still a significant gap between requirement specifications and software artifacts. We are still missing a tool that can extract meanings from requirements. Researchers can extract models from text using NLP techniques. Many researchers have also shown that it is possible to create UML models from free-text requirements. Researchers should pursue the automatic generation of source code from domain models in the future.

4.2 Summary

In this thesis, I went over how we may complement software engineering with

المنارة للاستشارات
www.manaraa.com

ontologies, and tackle a variety of challenges encountered in the software development cycle. I provided a review on current status of semantic software engineering and went over the advantages of applying ontologies in the various tasks in the software development cycle. I presented a number of ontologies that were designed to improve each phase in the waterfall model. I analyzed the usage of the different approaches proposed to facilitate tasks in the development of software engineering. I introduced an application of semantic annotation by presenting a tool that guides users in the capture of relevant information. This information allows the tool to then create an ontology for sharing and retrieval at a later time. The prototype is still a work-in-progress, but I believe the preliminary result has shown benefits in its usage.

I also pointed out areas that may be improved with the help of semantic technologies, and suggested future research directions in semantic software engineering.

المنارة للاستشارات
www.manaraa.com

References

[1] T. R. Gruber. A translation approach to portable ontologies. Knowledge Acquisition, 5(2):199-220, 1993.

[2] What is meta-modeling? http://infogrid.org/trac/wiki/Reference/WhatIsMetaModeling

[3] H. Kattenstroth, W. May, and F. Schenk. Combining OWL with F-Logic Rules and Defaults. In Proc. ALPSWS 2007, pp. 60-75, 2007.

[4] SPARQL http://www.w3.org/TR/rdf-sparql-query

[5] E. Sirin and B. Parsia. SPARQL-DL: SPARQL Query for OWLDL. In Christine Golbreich, Aditya Kalyanpur, and Bijan Parsia, editors, 3rd OWL Experiences and Directions Workshop (OWLED 2007), volume 258. CEUR Proceedings, 2007.

[6] M.J. O'Connor, A.K Das. SQWRL: a query language for OWL. OWL: Experiences and Directions (OWLED), Fifth International Workshop, Chantilly, VA, 2009.

[7] Protege http://protege.stanford.edu/

[8] Apache Jena http://incubator.apache.org/jena/

[9] E. Sirin, B. Parsia, B.C. Grau, A. Kalyanpur, Y. Katz. Pellet: A Practical OWL-DL Reasoner, Tech. Rep. CS 4766, University of Maryland, College Park, 2005.

[10] Hermit OWL Reasoner http://hermit-reasoner.com/

[11] FaCT++ http://owl.man.ac.uk/factplusplus/

[12] T. Gruber. Ontology. Entry in the Encyclopedia of Database Systems, Ling Liu and M. Tamer Özsu (Eds.), Springer-Verlag, to appear in 2008.

[13] D.M. Pisanelli, A. Gangemi, G. Steve. Ontologies and Information Systems: the Marriage of the Century?. In Proceedings of LYEE Workshop, Paris, 2002.

[14] P. Zave. Classification of research efforts in requirements engineering. ACM Comp. Sur., 29(4):315–321, 1997.

[15] S. J. Kormer and T. Brumm. RESI - A Natural Language Specification Improver. International Conference on Semantic Computing, 0:1–8, 2009.

[16] T. Riechert, K. Lauenroth, J. Lehmann, S. Auer. Towards Semantic based Requirements Engineering. Proceedings of the 7th International Conference on Knowledge Management, 144-151, 2007.

المنارة للاستشارات

www.manaraa.com

[17] G. Dobson and P. Sawyer. Revisiting Ontology-Based Requirements Engineering in the age of the Semantic Web. International Seminar on "Dependable Requirements Engineering of Computerised Systems at NPPs", Institute for Energy Technology (IFE), Halden, 2006.

[18] T. H. A. Balushi, P. R. F. Sampaio, D. Dabhi, P. Loucopoulos. ElicitO: A quality ontology-guided NFR elicitation tool. In Requirements Engineering: Foundation for Software Quality, pages 306-319. Springer Berlin / Heidelberg, 2007.

[19] P. Soffer, B. Golany, D. Dori, Y. Wand. Modeling off the-shelf information systems requirements: an ontological approach, Requirements Eng. 6, 183–199, 2001.

[20] J.C. Caralt, J.W. Kim, "Ontology Driven Requirements Query," hicss, pp.197c, 40th Annual Hawaii International Conference on System Sciences (HICSS'07), 2007.

[21] M. Shibaoka, H. Kaiya, and M. Saeki. GOORE: Goal-Oriented and Ontology Driven Requirements Elicitation Method. In Advances in Conceptual Modeling - Foundations and Applications, pages 225-234, Auckland, New Zealand, Nov. 2007. Springer. LNCS 4802.

[22] S.W. Lee and R.A. Gandhi. Ontology-based Active Requirements Engineering Framework. Proceedings of 12th Asia-Pacific Software Engineering Conf. (APSEC '05), Taiwan, IEEE CS Press, 2005.

[23] B. Decker, E. Ras, J. Rech, B. Klein, C. Hoecht. Self-organized reuse of software engineering knowledge supported by semantic wikis, in: Proceedings of the Workshop on Semantic Web Enabled Software Engineering (SWESE), ISWC, Galway, Ireland, 2005.

[24] V.S. Sharma, S. Sarkar, K. Verma, A. Panayappan, A. Kass. Extracting High-Level Functional Design from Software Requirements. APSEC, 35-42, 2009.

[25] K. Czarnecki, C.H.P. Kim, K.T. Kalleberg: Feature Models are Views on Ontologies. SPLC, 41-51, 2006.

[26] S. J. Korner and M. Landhaußer. Semantic Enriching of Natural Language Texts with Automatic Thematic Role Annotation. NLDB 2010, June 2010.

[27] S.J. Korner and T. Brumm. RESI - A Natural Language Specification Improver. International Conference on Semantic Computing, 0:1-8, 2009.

[28] A. Eberhart. Automatic Generation of Java/SQL Based Inference Engines from RDF Schema and RuleML. International Semantic Web Conference, 102-116, 2002.

المنارة للاستشارات

www.manaraa.com

[29] A. Kalyanpur, D. Jim´enez Pastor, S. Battle, and J. Padget. Automatic mapping of OWL ontologies into Java. In 16th International Conference on Software Engineering and Knowledge Engineering (SEKE), Banff, Canada, 2004

[30] G. Stevenson, S. Dobson. Sapphire: Generating java runtime artefacts from owl ontologies. In: CAiSE Workshops. pp. 425–436, 2011.

[31] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995.

[32] H. Kampffmeyer, S. Zschaler, G. Engels, B. Opdyke, D.C. Schmidt, F. Weil. Finding the pattern you need: The design pattern intent ontology. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, 211-225. Springer, Heidelberg, 2007.

[33] Z. Zygkostiotis, D. Dranidis, D. Kourtesis, Semantic Annotation, Publication and Discovery of Java Software Components: An Integrated Approach. In Proceedings of the 2nd Workshop on Artificial Intelligence Techniques in Software Engineering, 5th IFIP Conference on Artificial Intelligence Applications and Innovations, 2009.

[34] K. Kannan and B. Srivastava. Promoting reuse via extraction of domain concepts and service abstractions from design diagrams. Proceedings of SCC, 2008.

[35] H. Happel, A. Korthaus, S. Seedorf, P. Tomczyk. KOntoR: An Ontology-enabled Approach to Software Reuse. SEKE, 349-354, 2006.

[36] V. Sugumaran and V.C. Storey. A Semantic-Based Approach to Component Retrieval. The DATA BASE for Advances in Information Systems 34, 8–24 Quarterly publication of the Special Interest Group on Management Information Systems of the Association for Computing Machinery (ACM-SIGMIS), 2003.

[37] T. Emerson, J. Reising, and H. Britten-Austin. Workload and Situation Awareness in Future Aircraft. SAE Technical Paper 871803, 1987.

[38] C. J. Matheus, K. Baclawski, M. M. Kokar, J. Letkowski: Using SWRL and OWL to Capture Domain Knowledge for a Situation Awareness Application Applied to a Supply Logistics Scenario. RuleML , 130-144, 2005.

[39] N. Baumgartner, W. Retschitzegger, W. Schwinger. A Software Architecture for Ontology-Driven Situation Awareness, ACM SA Conference, 2008.

[40] Y. Luo. Improving Software Quality Using An Ontology-based Approach. LAP Lambert Academic, 2010.

المنارة للاستشارات

www.manaraa.com

[41] L. Yu, J. Zhou, Y. Yi, P. Li, Q. Wang. Ontology Model-Based Static Analysis on Java Programs, The 32nd Annual IEEE International Computer Software and Applications Conference, 92-99, 2008.

[42] A. Rauf, S. Anwar, M. Ramzan, A.A. Shahid. Ontology Driven Semantic Annotation Based GUI Testing. IEEE International Conference on Emerging Technologies 2010 (ICET 2010) Islamabad, Pakistan, October 18-19, 2010.

[44] V. H. Nasser, W. Du, and D.MacIsaac. Knowledge-based Software Test Generation, in Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering, Boston, July, 2009.

[46] D.C. Nguyen, A. Perini, P. Tonella. eCAT: a tool for automating test cases generation and execution in testing multi-agent systems. AAMAS, 1669-1670, 2008.

[47] M. Serhatli, F.N. Alpaslan. An ontology based question answering system on software test document domain. World Academy of Science, Engineering and Technology, 2009.

[48] IEEE Std. 610.12, Standard Glossary of Software Engineering Terminology. IEEE Computer Society Press, Los Alamitos, CA, 1990.

[49] B.A. Kitchenham, G.H. Travassos, A. Von Mayrhauser, F. Niessink, N.F. Schniedewind, J. Singer, S. Takado, R. Vehvilainen, H. YANG. Towards an Ontology of Software Maintenance, Journal of Software Maintenance: Research and Practice, 11(6):365-389, 1999.

[50] F. Ruiz, A. Vizcaíno, M. Piattini, and F. García, An ontology for the management of software maintenance projects. International Journal of Software Engineering and Knowledge Engineering, 14(3):323–349, 2004.

[51] M.G. Dias, N. Anquetil, K.M. De Oliveira. Organizing the Knowledge Used in Software Maintenance. Journal of Universal Computer Science, 9(7): 641–658, 2003.

[52] W. Meng, J. Rilling, Y. Zhang, R. Witte, P. Charland, “An Ontological Software Comprehension Process Model”, In Proc. of the 3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering, Italy, 2006.

[53] S.L. Abebe, P. Tonella. Natural Language Parsing of Program Element Names for Concept Extraction. ICPC, 156-159, 2010.

[54] A. Hass. What is Configuration Management from Software Configuration Management: Principles and Practices; Addison-Wesley, December 2002.

[55] H.H. Shahri, J. Hendler, A. Porter. Software Configuration Management Using Ontologies. Proceedings of the 3rd International Workshop on Semantic Web Enabled

المنارة للاستشارات
www.manaraa.com

Software Engineering at the 4th European Semantic Web Conference (ESWC'07), Innsbruck, Austria, June 6-7, 2007.

[56] D. Hyland-Wood, D. Carrington, and S. Kaplan. Toward a Software Maintenance Methodology Using Semantic Web Techniques. Proceedings of Second International IEEE Workshop Software Evolvability, 23-30, Sept. 2006.

[57] J. Tappolet, C. Kiefer, A. Bernstein. Semantic web enabled software analysis. Web Semantic. Sci. Serv. Agents World Wide Web 8, 225–240, 2010.

[58] FuXi http://code.google.com/p/fuxi/

[59] R. Witte, Y. Zhang, and J. Rilling. Empowering Software Maintainers with Semantic Web Technologies. 4th European Semantic Web Conference, June 3-7, 2007, Innsbruck, Austria. Springer LNCS 4519, 37-52, 2007.

[60] A. Ambrosio, D. Santos, F. Lucena, J. Silva. Software engineering documentation: an ontology-based approach. Proceedings of the 10th Brazilian Symposium on Multimedia and the Web 2nd Latin American Web Congress, Oct. 2004.

[61] N. Khamis, R. Witte, J. Rilling. Automatic Quality Assessment of Source Code Comments: The JavadocMiner. 15th International Conference on Applications of Natural Language to Information Systems, June 23–25, Cardiff, UK. Springer LNCS 6177, 68–79, 2010.

[62] S. Fenz and E. Weippl. Ontology based IT-security planning. Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing PRDC '06. IEEE Computer Society, 389-390, 2006.

[63] J. Undercoffer, A. Joshi, J. Pinkston. Modeling Computer Attacks: An Ontology for Intrusion Detection. in The Sixth International Symposium on Recent Advances in Intrusion Detection. Springer, 2003.

[64] A Course in Black Box Software Testing, Cem Kaner 2004

[65] wxPython http://www.wxpython.org

[66] Xunit Test Patterns http://xunitpatterns.com/

[67] Web Ontology Language http://www.w3.org/TR/owl-features/

[68] L. Moulton. "Semantic Software Technologies: Landscape of High Value Applications for the Enterprise." http://www.expertsystem.net/documenti/pdf_eng/technology/semanticsoftwaretechnologies_gilbane2010.pdf. 5 Aug. 2010. Web. 1 Mar. 2012

المنارة للاستشارات
www.manaraa.com

[69] I. Sommerville and P. Sawyer. Requirements Engineering: A Good Practice Guide. Wiley, Apr. 1997.

المنارة للاستشارات
www.manaraa.com

Appendix A

```
#!/usr/bin/env python

def partition(list, start, end):
	pivot = list[end]
	bottom = start-1
	top = end

	done = 0
	while not done:

		while not done:
			bottom = bottom+1

			if bottom == top:
				done = 1
				break

			if list[bottom] > pivot:
				list[top] = list[bottom]
				break

		while not done:
			top = top-1

			if top == bottom:
				done = 1
				break

			if list[top] < pivot:
				list[bottom] = list[top]
				break

	list[top] = pivot
	return top

def quicksort(list, start, end):
	if start < end:
		split = partition(list, start, end)
		quicksort(list, start, split-1)
		quicksort(list, split+1, end)
	else:
		return
```


المنارة للاستشارات
www.manaraa.com

```
if __name__=="__main__":
        import sys
        list = map(int,sys.argv[1:])
        start = 0
        end = len(list)-1
        quicksort(list,start,end)
        import string
        print string.join(map(str,list))
```


المنارة للاستشارات
www.manaraa.com

Appendix B

```
import quicksort
import unittest
import random

class TestQuickSort(unittest.TestCase):
	def setUp(self):
		self.list = range(100)
		random.shuffle(self.list)

	def testNormalInput(self):
		'''check if normal input works'''
		quicksort.quicksort(self.list, 0, len(self.list)-1)
		log = self.assertEqual(self.list, range(100))
		print "testNormalInput : ", log

	def testOneNumber(self):
		input = self.list[0]
		quicksort.quicksort(input, 0, 0)
		log = self.assertEqual(input, self.list[0])
		print "testOneNumber : ", log

	def testNegativeInput(self):
		counter = 0
		for n in self.list:
			n = n * -1
			self.list[counter] = n
			counter = counter + 1
		input = self.list
		self.list.sort()
		quicksort.quicksort(input, 0, len(input)-1)
		log = self.assertEqual(self.list, input)
		print "testNegativeInput : ", log

	def testNonIntegers(self):
		alist = ['a', 'b', 'c', 'd', 'e']
		input = alist
		random.shuffle(input)
		quicksort.quicksort(input, 0, len(input)-1)
		log = self.assertEqual(alist, input)
		print "testNonIntegers : ", log

	def testSameInput(self):
		counter = 0
```


المنارة للاستشارات
www.manaraa.com

```
        for n in self.list:
            self.list[counter] = self.list[0]
            counter = counter+1
        input = self.list
        quicksort.quicksort(input, 0, len(input)-1)
        log = self.assertEqual(input, self.list)
        print "testSameInput : ", log

    def testSortedList(self):
        self.list.sort()
        quicksort.quicksort(self.list, 0, len(self.list)-1)
        log = self.assertEqual(self.list, range(5))
        print "testSortedList : ", log

    def testReversedList(self):
        self.list.sort(reverse=True)
        quicksort.quicksort(self.list, 0, len(self.list)-1)
        log = self.assertEqual(self.list, range(100))
        print "testReversedList : ", log

def PrintResult(log):
    f = open('result.log', 'w')
    f.write(log)

if __name__ == '__main__':
    import sys
    suite = unittest.TestLoader().loadTestsFromTestCase(TestQuickSort)
    unittest.TextTestRunner(stream=sys.stdout, verbosity=2).run(suite)
```


المنارة للاستشارات
www.manaraa.com